CS420 – Lecture 17

Marc Snir

Fall 2018

Principal Component Analysis
Partial

Covariance matrix

- $\vec{v}_0, \ldots, \vec{v}_{m-1}$ are *feature vectors* (e.g., pixels of an image); $\vec{v}_i = (v_0^{(i)}, \ldots, v_{n-1}^{(i)})$. m is the number of observations, the n is number of features per observation. Both can be large and typically, $m \gg n$.
- Mean: $\vec{\mu} = \frac{1}{m} \sum_{i=0}^{m-1} \vec{v}_i$.
- Covariance matrix: $C_{jj'} = \frac{1}{m} \sum_{i=0}^{m-1} (v_j^{(i)} - \mu_j)(v_{j'}^{(i)} - \mu_{j'})$. $n \times n$ matrix.

PCA achieved by diagonalizing the covariance matrix.

Assume we have already computed mean μ ($\approx nm$ operations, easy to parallelize); and computed $\bar{v}_i = \vec{v}_i - \mu$ (again $\approx mn$ operations, easy to parallelize).

We focus on the computation of the covariance matrix $C_{jj'} = \frac{1}{m} \sum_{i=0}^{m-1} \bar{v}_j^{(i)} \bar{v}_{j'}^{(i)}$; $\approx mn^2$ operations.

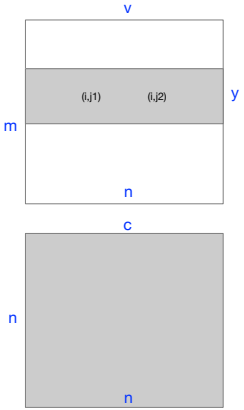
```
for (j1=0;j1<n;j1++)
 for (j2=0; j2<=j1;j2++)
  {
  for (i=0; i<m; i++)
   c[j1][j2] +=v[i][j1]*v[i][j2];
  c[j1][j2] /= m;
 }
```

Taking advantage of symmetry: $C_{jj'} = C_{j'j}$

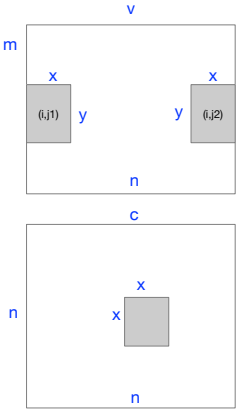
Sequential code - tiling

Two possible tiling strategies:

One tile

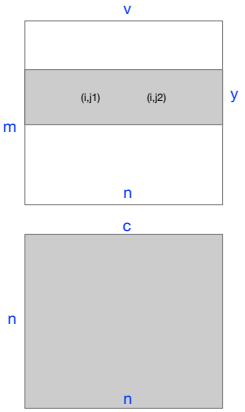


Two tiles



One tile

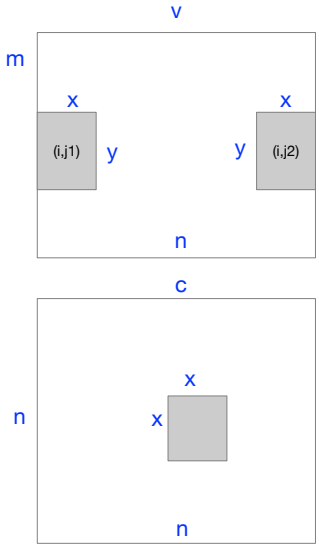
One tile



- Need to keep in cache $\approx n^2/2 + n$ elements
- If can fit so many elements in cache, then tiling is not necessary
- Total number of cache misses is proportional to $nm + n^2$ – This is optimal

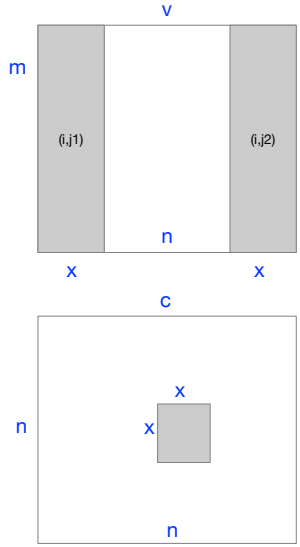
Covarience matrix does not fit in cache

One tile



- Need to keep in cache $\approx x^2 + 2x$ elements
- Working on 2 tiles takes $\approx x^2 y$ operations
- Number of cache misses is proportional to $\approx x^2 + 2xy$
- Ratio between misses and work is $\frac{x^2+2xy}{x^2y} = \frac{1}{y} + \frac{2}{x}$. need to pick x and y as large as possible.
- $y = m$, $x \approx \sqrt{\text{cache\_size}}$

Two tiles



- $x \approx \sqrt{\texttt{cache\_size}}$
- Number of cache misses on matrix c is proportional to n^2.
- Number of cache misses on matrix v is proportional to $\approx \frac{1}{2}(\frac{n}{x})^2 mx = \frac{mn^2}{2x} \approx \frac{mn^2}{2\sqrt{\texttt{cache\_size}}}$
- Total of $\approx n^2(1 + \frac{m}{2\sqrt{\texttt{cache\_size}}})$
- (Constant of proportionality is number of entries per cache line).

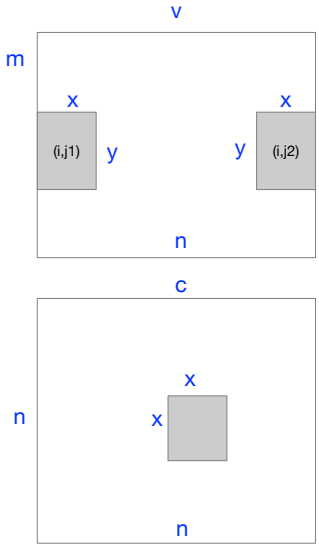
Tiled code

$T \approx \sqrt{\text{cache\_size}}$

```
for (J1=0;J1<n;J1+=T) {
 last1 = J1+T < n ? j1+T:n;
 for (J2=0; J2<=J1;J2+=T) {
  last2 = J2+T <last1 ? J2+T:last1;
  for (i=0;i<m;i++)
   for (j1=J1;j1<last1;j1++)
    for (j2=J2;j2<last2;j2++) {
     c[j1][j2] +=v[i][j1]*v[i][j2];
    }
   }
  }
```

Correct value of `T` needs to be determined by empirical search.
Computation of `last1` and `last2` can be avoided by splitting the loops into two parts.

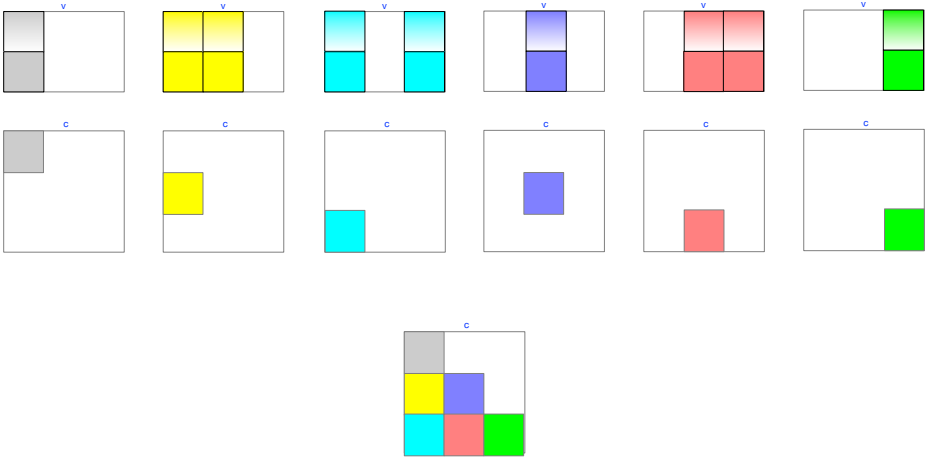
Compute on GPU



- GPU memory is much smaller than CPU memory
- Data needs to be transfered from CPU memory to GPU memory ahead of kernel computation
  - (As distinct from transfer to cache that happens concurrently with computation)
- Need to keep in GPU memory $\approx x^2 + 2xy$ elements
- Can keep $x \times x$ submatrix of c for multiple phases
- Each phase requires transfer of $\approx 2xy$ elements and $\approx x^2 y$ computations

- Need to maximize ratio $\frac{x^2 y}{2xy} = x/2$ under the constraint $x^2 + 2xy < \texttt{GPU\_mem\_size}$.
- Take $y = 1$ and $x \approx \sqrt{\texttt{GPU\_mem\_size}}$
- More accurate analysis will take into account time to spawn kernel ($\approx 10\mu s$) – and will lead to $x \gg y > 1$.

Tile Computation



Target map constraint

- Problem: Can transfer only contiguous data from CPU to GPU and back
- Possible solution: First copy to contiguous buffer, next copy

functions to gather/scatter submatrices

```
  ...
void flatten(int m, int n, float a[m][n],
   int i, int j, int ilen, int jlen, float* p) {
 int r,s;
 for (r=i; r<i+ilen; i++)
  for (s=j; s<j+jlen; j++)
   *p++ = a[r][s];
 }

void unflatten(int m, int n, float a[m][n],
      int i, int j, int ilen, int jlen, float* p) {
 int r,s;
 for (r=i; r<i+ilen; i++)
  for (s=j; s<j+jlen; j++)
   a[r][s] = *p++;
}
```

functions to compute convolution tile

```
#pragma omp declare target
 void tile2(int x, int y, float* v1, float* v2, float* c) {
  /* disjoint tiles */
  int r1,r2,s;
  for (r1=0; r1<x; r1++)
   for (r2=0; r2<x; r2++)
    for(s=0;s<y;s++)
     *(c+r1*x+r2) += *(v1+s*x+r1)* *(v2+s*x+r2)
 }

void tile1(int x, int y, float* v, float* c) {
  /* same tile */
  int r1,r2,s;
  for (r1=0; r1<x; r1++)
  for (r2=0; r2<=r1; r2++)
  for(s=0;s<ilen;s++)
  *(c+r1*x+r2) += *(v+s*x+r1)* *(v+s*x+r2)
}
#pragma omp end declare target
```

Main Code

$T \approx \sqrt{\text{cache\_size}}$; $1 < t \ll T$.

```
for (J1=0;J1<n;J1+=T) {
 last1 = J1+T < n ? j1+T:n;
 for (J2=0; J2<=J1;J2+=T) {
  last2 = J2+T < last1 ? J2+T:last1;
  flatten(n,n,c[n][n],J1,J2,T,T,ctemp);
  #pragma omp target entry data map(to: ctemp[:T*T], T),
   for(I=0;I<m;I+=t) {
    last3 = I+t<m ? I+t:m;
    if (J1=J2) {
     flatten(m,n,v[m][n],I,J1,t,T, v1temp);
     #pragma target map(to:v1temp[:T*t],t)
      tile1(T, t, v1temp, ctemp);
     }
    else {
     flatten(m,n,v[m][n],I,J1,t,T, v1temp);
     flatten(m,n,v[m][n],I,J1,t,T, v2temp);
     #pragma target map(to:v1temp[:T*t], v2temp[:T*t], t)
       tile2(T, t, v1temp, v2temp, ctemp);
     }
```

```
    }
   #pragma omp target exit data map(from: ctemp[:T*T])
   unflatten(n,n,c[n][n],J1,J2,T,T,ctemp);
  }
}
for (i=0;i<n;i++)
 for (j=0;j=n;j++)
  c[j1][j2] /= m;
```